Problemas $\mathcal{NP}$-Completo e Algoritmos Aproximados*

Última alteração: 28 de Setembro de 2010

*Transparências elaboradas por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

Conteúdo do Capítulo

Introdução

- Problemas intratáveis ou difíceis são comuns na natureza e nas áreas do conhecimento.
- Problemas “fáceis”: resolvidos por algoritmos polinomiais.
- Problemas “difíceis”: somente possuem algoritmos exponenciais para resolvê-los.
- A complexidade de tempo da maioria dos problemas é polinomial ou exponencial.
- **Polinomial**: função de complexidade é $O(p(n))$, onde $p(n)$ é um polinômio.
  - Ex.: algoritmos com pesquisa binária ($O(\log n)$), pesquisa sequencial ($O(n)$), ordenação por inserção ($O(n^2)$), e multiplicação de matrizes ($O(n^3)$).
- **Exponencial**: função de complexidade é $O(c^n)$, $c > 1$.
  - Ex.: **problema do caixeiro viajante** (PCV) ($O(n!)$).
  - Mesmo problemas de tamanho pequeno a moderado não podem ser resolvidos por algoritmos não-polinomiais.

Problemas $\mathcal{NP}$-Completo

- A teoria de complexidade a ser apresentada não mostra como obter algoritmos polinomiais para problemas que demandam algoritmos exponenciais, nem afirma que não existem.
- É possível mostrar que os problemas para os quais não há algoritmo polinomial conhecido são computacionalmente relacionados.
- Formam a classe conhecida como $\mathcal{NP}$.
- Propriedade: um problema da classe $\mathcal{NP}$ poderá ser resolvido em tempo polinomial se e somente se todos os outros problemas em $\mathcal{NP}$ também puderem.
- Este fato é um indício forte de que dificilmente alguém será capaz de encontrar um algoritmo eficiente para um problema da classe $\mathcal{NP}$.

Coloração de um Grafo - Otimização de Compiladores

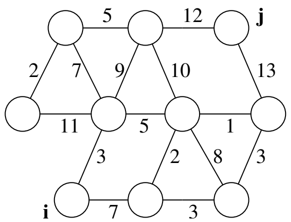
- Escalonar o uso de um número finito de registradores (idealmente com o número mínimo).
- No trecho de programa a ser otimizado, cada variável tem intervalos de tempo em que seu valor tem de permanecer inalterado, como depois de inicializada e antes do uso final.
- Variáveis com interseção nos tempos de vida útil não podem ocupar o mesmo registrador.
- Modelagem por grafo: vértices representam variáveis e cada aresta liga duas variáveis que possuem interseção nos tempos de vida.
- Coloração dos vértices: atibui cada variável a um agrupamento (ou classe). Duas variáveis com a mesma cor não colidem, podendo assim ser atribuídas ao mesmo registrador.

Coloração de um Grafo

- Em um grafo $G = (V, A)$, mapear $C : V \leftarrow S$, sendo S um conjunto finito de cores tal que se $\overline{vw} \in A$ então $c(v) \neq c(w)$ (vértices adjacentes possuem cores distintas).
- O número cromático $X(G)$ de G é o menor número de cores necessário para colorir G, isto é, o menor k para o qual existe uma coloração C para G e $|C(V)| = k$.
- O problema é produzir uma coloração ótima, que é a que usa apenas $X(G)$ cores.
- Formulação do tipo "sim/não": dados G e um inteiro positivo k, existe uma coloração de G usando k cores?
  - *Fácil*: $k = 2$.
  - *Difícil*: $k > 2$.
- Aplicação: modelar problemas de agrupamento (*clustering*) e de horário (*scheduling*).

Caminho em um Grafo

- Considere um grafo com peso nas arestas, dois vértices i, j e um inteiro $k > 0$.



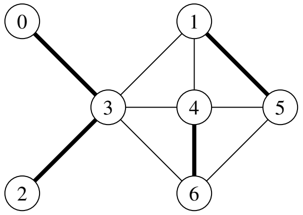
- *Fácil:* Existe um caminho de i até j com peso $\leq k$?.
  - Há um algoritmo eficiente com complexidade de tempo $O(A \log V)$, sendo A o número de arestas e V o número de vértices (algoritmo de Dijkstra).
- *Difícil:* Existe um caminho de i até j com peso $\geq k$?
  - Não existe algoritmo eficiente. É equivalente ao PCV em termos de complexidade.

Classe $\mathcal{NP}$ - Problemas "Sim/Não"

- Para o estudo teórico da complexidade de algoritmos considera-se problemas cujo resultado da computação seja "sim" ou "não".
- Versão do Problema do Caixeiro Viajante (PCV) cujo resultado é do tipo "sim/não":
  - Dados: uma constante k, um conjunto de cidades $C = \{c_1, c_2, \cdots, c_n\}$ e uma distância $d(c_i, c_j)$ para cada par de cidades $c_i, c_j \in C$.
  - Questão: Existe um "roteiro" para todas as cidades em C cujo comprimento total seja menor ou igual a k?
- Característica da classe $\mathcal{NP}$: problemas "sim/não" para os quais uma dada solução pode ser verificada facilmente.
- A solução pode ser muito difícil ou impossível de ser obtida, mas uma vez conhecida ela pode ser verificada em tempo polinomial.

Cobertura de Arestas

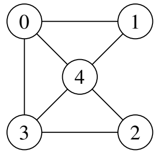
- Uma **cobertura de arestas** de um grafo $G = (V, A)$ é um subconjunto $A' \subset A$ de k arestas tal que todo $v \in V$ é parte de pelo menos uma aresta de A'.
- O conjunto resposta para $k = 4$ é $A' = \{(0, 3), (2, 3), (4, 6), (1, 5)\}$.



- Uma **cobertura de vértices** é um subconjunto $V' \subset V$ tal que se $(u, v) \in A$ então $u \in V'$ ou $v \in V'$, isto é, cada aresta do grafo é incidente em um dos vértices de V'.
- Na figura, o conjunto resposta é $V' = \{3, 4, 5\}$, para $k = 3$.
- Dados um grafo e um inteiro $k > 0$
  - *Fácil*: há uma cobertura de arestas $\leq k$?.
  - *Difícil*: há uma cobertura de vértices $\leq k$?

Ciclo de Hamilton

- **Ciclo de Hamilton**: **ciclo simples** (passa por todos os vértices uma única vez).
- **Caminho de Hamilton**: **caminho simples** (passa por todos os vértices uma única vez).
- Exemplo de ciclo de Hamilton: 0 1 4 2 3 0. Exemplo de caminho de Hamilton: 0 1 4 2 3.



- Existe um ciclo de Hamilton no grafo G?
  - *Fácil*: Grafos com grau máximo = 2 (vértices com no máximo duas arestas incidentes).
  - *Difícil*: Grafos com grau > 2.
- É um caso especial do PCV. Pares de vértices com uma aresta entre eles tem distância 1 e pares de vértices sem aresta entre eles têm distância infinita.

Coloração de um Grafo - Problema de Horário

- Suponha que os exames finais de um curso tenham de ser realizados em uma única semana.
- Disciplinas com alunos de cursos diferentes devem ter seus exames marcados em horários diferentes.
- Dadas uma lista de todos os cursos e outra lista de todas as disciplinas cujos exames não podem ser marcados no mesmo horário, o problema em questão pode ser modelado como um problema de coloração de grafos.

Coloração de um Grafo - Otimização de Compiladores

- Evidentemente, não existe conflito se cada vértice for colorido com uma cor distinta.
- O objetivo porém é encontrar uma coloração usando o mínimo de cores (computadores têm um número limitado de registradores).
- **Número cromático**: menor número de cores suficientes para colorir um grafo.

Máquina Não-Determinista

- Uma máquina capaz de executar a função *escolhe* admite a capacidade de **computação não-determinista**.
- Uma máquina não-determinista é capaz de produzir cópias de si mesma quando diante de duas ou mais alternativas, e continuar a computação independentemente para cada alternativa.
- A máquina não-determinista que acabamos de definir não existe na prática, mas ainda assim fornece fortes evidências de que certos problemas não podem ser resolvidos por algoritmos deterministas em tempo polinomial, conforme mostrado na definição da classe $\mathcal{NP}$-completo à frente.

Comandos *sucesso* e *insucesso*

- Algoritmos não-deterministas utilizam também dois comandos, a saber:
  - *insucesso*: indica término sem sucesso.
  - *sucesso*: indica término com sucesso.
- Os comandos *insucesso* e *sucesso* são usados para definir uma execução do algoritmo.
- Esses comandos são equivalentes a um comando de parada de um algoritmo determinista.
- Os comandos *insucesso* e *sucesso* também têm complexidade de tempo $O(1)$.

Função *escolhe(C)*

- Algoritmos não-deterministas utilizam uma função *escolhe(C)*, que escolhe um dos elementos do conjunto C de forma arbitrária.
- O comando de atribuição X $\leftarrow$ *escolhe* (1:n) pode resultar na atribuição a X de qualquer dos inteiros no intervalo $[1, n]$.
- A complexidade de tempo para cada chamada da função *escolhe* é $O(1)$.
- Neste caso, não existe nenhuma regra especificando como a escolha é realizada.
- Se um conjunto de possibilidades levam a uma resposta, este conjunto é escolhido sempre e o algoritmo terminará com sucesso.
- Por outro lado, um algoritmo não-determinista termina sem sucesso se e somente se não há um conjunto de escolhas que indica sucesso.

Algoritmos Não-Deterministas

- **Algoritmos deterministas**: o resultado de cada operação é definido de forma única.
- Em um arcabouço teórico, é possível remover essa restrição.
- Apesar de parecer irreal, este é um conceito importante e geralmente utilizado para definir a classe $\mathcal{NP}$.
- Neste caso, os algoritmos podem conter operações cujo resultado não é definido de forma única.
- **Algorimo não-determinista**: capaz de escolher uma dentre as várias alternativas possíveis a cada passo.
- Algoritmos não-deterministas contêm operações cujo resultado não é unicamente definido, ainda que limitado a um conjunto especificado de possibilidades.

Problema da Satisfabilidade

- O algoritmo AvalND(E,n) verifica se uma expressão E na forma normal conjuntiva, com variáveis $x_i, 1 \le i \le n$, é *satisfatível*.

```
procedure AvalND (E, n);
begin
  for i := 1 to n do
    begin
    xi ← escolhe(true, false);
    if E(x1, x2, ···, xn) = true then sucesso else insucesso;
    end;
end;
```


Problema da Satisfabilidade

- Considere um conjunto de **variáveis booleanas** $x_1, x_2, \cdots, x_n$, que podem assumir valores lógicos *verdadeiro* ou *falso*.
- A negação de x_i é representada por $\overline{x_i}$.
- Expressão booleana: variáveis booleanas e operações **ou** ($\vee$) e **e** ($\wedge$). (também chamadas respectivamente de adição e multiplicação).
- Uma expressão booleana E contendo um produto de adições de variáveis booleanas é dita estar na **forma normal conjuntiva**.
- Dada E na forma normal conjuntiva, com variáveis $x_i, 1 \le i \le n$, existe uma atribuição de valores verdadeiro ou falso às variáveis que torne E verdadeira ("satisfaça")?
- $E_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3} \vee x_2) \wedge (x_3)$ é *satisfatível* ($x_1 = F$, $x_2 = V$, $x_3 = V$).
- A expressão $E_2 = x_1 \wedge \overline{x_1}$ não é *satisfatível*.

Ordenação Não-Determinista

- Ordenar um conjunto $A[1:n]$ contendo n inteiros positivos, $n \ge 1$.

```
procedure OrdenaND (A, 1, n);
begin
  for i := 1 to n do B[i] := 0;
  for i := 1 to n do
    begin
    j ← escolhe(A,1,n);
    if B[j] = 0 then B[j] := A[i]
    else insucesso;
    end;
end;
```

  - Um vetor auxiliar B[1:n] é utilizado. Ao final, B contém o conjunto em ordem crescente.
  - A posição correta em B de cada inteiro de A é obtida de forma não-determinista pela função escolhe.

- Em seguida, o comando de decisão verifica se a posição B[j] ainda não foi utilizada.
- A complexidade é $O(n)$. (Para um algoritmo determinista a complexidade é $\Omega(n \log n)$)

Pesquisa Não-Determinista

- Pesquisar o elemento x em um conjunto de elementos $A[1:n]$, $n \ge 1$.

```
procedure PesquisaND (A, 1, n);
begin
  j ← escolhe(A,1,n)
  if A[j] = x then sucesso else insucesso;
end;
```

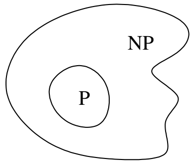
- Determina um índice j tal que $A[j] = x$ para um término com sucesso ou então insucesso quando x não está presente em A.
- O algoritmo tem complexidade não-determinista $O(1)$.
- Para um algoritmo determinista a complexidade é $\Omega(n)$.

$\mathcal{NP} \supset \mathcal{P}$ ou $\mathcal{NP} = \mathcal{P}$? - Conseqüências

- Muitos problemas práticos em $\mathcal{NP}$ podem ou não pertencer a $\mathcal{P}$ (não conhecemos nenhum algoritmo determinista eficiente para eles).
- Se conseguirmos provar que um problema não pertence a $\mathcal{P}$, então não precisamos procurar por uma solução eficiente para ele.
- Como não existe tal prova, sempre há esperança de que alguém descubra um algoritmo eficiente.
- Quase ninguém acredita que $\mathcal{NP} = \mathcal{P}$.
- Existe um esforço considerável para provar o contrário, mas a questão continua em aberto!

Existe Diferença entre $\mathcal{P}$ e $\mathcal{NP}$?

- $\mathcal{P} \subseteq \mathcal{NP}$, pois algoritmos deterministas são um caso especial dos não-deterministas.
- A questão é se $\mathcal{P} = \mathcal{NP}$ ou $\mathcal{P} \neq \mathcal{NP}$.
- Esse é o problema não resolvido mais famoso que existe na área de ciência da computação.
- Se existem algoritmos polinomiais deterministas para todos os problemas em $\mathcal{NP}$, então $\mathcal{P} = \mathcal{NP}$.
- Por outro lado, a prova de que $\mathcal{P} \neq \mathcal{NP}$ parece exigir técnicas ainda desconhecidas.
- Descrição tentativa do mundo $\mathcal{NP}$, em que a classe $\mathcal{P}$ está contida na classe $\mathcal{NP}$.



  – Acredita-se que $\mathcal{NP} \gg \mathcal{P}$, pois para muitos problemas em $\mathcal{NP}$, não existem algoritmos polinomiais conhecidos, nem um **limite inferior não-polinomial** provado.

Caracterização das Classes $\mathcal{P}$ e $\mathcal{NP}$

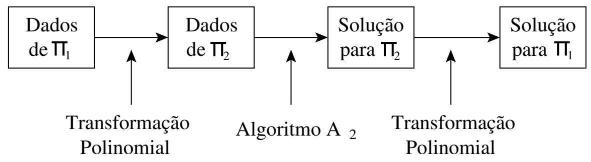
- $\mathcal{P}$: conjunto de todos os problemas que podem ser resolvidos por *algoritmos deterministas* em tempo *polinomial*.
- $\mathcal{NP}$: conjunto de todos os problemas que podem ser resolvidos por *algoritmos não-deterministas* em tempo *polinomial*.
- Para mostrar que um determinado problema está em $\mathcal{NP}$, basta apresentar um algoritmo não-determinista que execute em tempo polinomial para resolver o problema.
- Outra maneira é encontrar um algoritmo determinista polinomial para verificar que uma dada solução é válida.

Problema da Satisfabilidade

- O algoritmo obtém uma das 2^n atribuições possíveis de forma não-determinista em $O(n)$.
- Melhor algoritmo determinista: $O(2^n)$.
- Aplicação: definição de circuitos elétricos combinatórios que produzam valores lógicos como saída e sejam constituídos de portas lógicas **e**, **ou** e **não**.
- Neste caso, o mapeamento é direto, pois o circuito pode ser descrito por uma expressão lógica na forma normal conjuntiva.

Transformação Polinomial

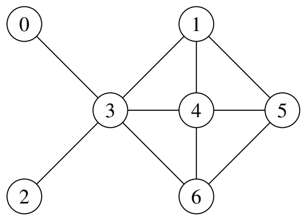
- Sejam Π_1 e Π_2 dois problemas "sim/não".
- Suponha que um algoritmo A_2 resolva Π_2.
- Se for possível transformar Π_1 em Π_2 e a solução de Π_2 em solução de Π_1, então A_2 pode ser utilizado para resolver Π_1.
- Se pudermos realizar as transformações nos dois sentidos em tempo polinomial, então Π_1 é *polinomialmente transformável* em Π_2.



- Esse conceito é importante para definir a classe $\mathcal{NP}$-completo.
- Para mostrar um exemplo de transformação polinomial, definiremos clique de um grafo e conjunto independente de vértices de um grafo.

Conjunto Independente de Vértices de um Grafo

- O conjunto independente de vértices de um grafo $G = (V, A)$ é constituído do subconjunto $V' \subseteq V$, tal que $v, w \in V' \Rightarrow (v, w) \notin A$.
- Todo par de vértices de V' é não adjacente (V' é um subgrafo totalmente desconectado).
- Exemplo de cardinalidade 4: $V' = \{0, 2, 1, 6\}$.

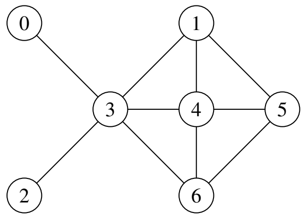


Conjunto Independente de Vértices - Aplicação

- Em problemas de dispersão é necessário encontrar grandes conjuntos independentes de vértices. Procura-se um conjunto de pontos mutuamente separados.
- Exemplo: identificar localizações para instalação de franquias.
- Duas localizações não podem estar perto o suficiente para competirem entre si.
- Solução: construir um grafo em que possíveis localizações são representadas por vértices, e arestas são criadas entre duas localizações que estão próximas o suficiente para interferir.
- O maior conjunto independente fornece o maior número de franquias que podem ser concedidas sem prejudicar as vendas.
- Em geral, cunjuntos independentes evitam conflitos entre elementos.

Clique de um grafo

- **Clique** de um grafo $G = (V, A)$ é constituído do subconjunto $V' \subseteq V$, tal que $v, w \in V' \Rightarrow (v, w) \in A$.
- Todo par de vértices de V' é adjacente (V' é um subgrafo completo).
- Exemplo de cardinalidade 3: $V' = \{3, 1, 4\}$.



Problemas $\mathcal{NP}$-Completo e $\mathcal{NP}$-Difícil

- Dois problemas Π_1 e Π_2 são **polinomialmente equivalentes** se e somente se $\Pi_1 \propto \Pi_2$ e $\Pi_2 \propto \Pi_1$.
- Exemplo: **problema da *satisfabilidade***. Se $SAT \propto \Pi_1$ e $\Pi_1 \propto \Pi_2$, então $SAT \propto \Pi_2$.
- Um problema Π é $\mathcal{NP}$**-difícil** se e somente se $SAT \propto \Pi$ (*satisfabilidade* é redutível a Π).
- Um problema de decisão Π é denominado $\mathcal{NP}$**-completo** quando:
  1. $\Pi \in NP$;.
  2. Todo problema de decisão $\Pi' \in \mathcal{NP}$-completo satisfaz $\Pi' \propto \Pi$.

Transformação Polinomial

- Se existe um algoritmo que resolve o conjunto independente em tempo polinomial, ele pode ser utilizado para resolver clique também em tempo polinomial.
- Diz-se que clique $\propto$ conjunto independente.
- Denota-se $\Pi_1 \propto \Pi_2$ para indicar que Π_1 é polinomialmente transformável em Π_2.
- A relação $\propto$ é transitiva ($\Pi_1 \propto \Pi_2$ e $\Pi_2 \propto \Pi_3 \Rightarrow \Pi_1 \propto \Pi_3$).

Transformação Polinomial

- Considere Π_1 o problema clique e Π_2 o problema conjunto independente de vértices.
- A instância I de clique consiste de um grafo $G = (V, A)$ e um inteiro $k > 0$.
- A instância $f(I)$ de conjunto independente pode ser obtida considerando-se o grafo complementar $\overline{G}$ de G e o mesmo inteiro k.
- $f(I)$ é uma transformação polinomial:
  1. $\overline{G}$ pode ser obtido a partir de G em tempo polinomial.
  2. G possui clique de tamanho $\geq k$ se e somente se $\overline{G}$ possui conjunto independente de vértices de tamanho $\geq k$.

Clique de um grafo - Aplicação

- O problema de identificar agrupamentos de objetos relacionados freqüentemente se reduz a encontrar grandes cliques em grafos.
- Exemplo: empresa de fabricação de peças por meio de injeção plástica que fornece para diversas outras empresas montadoras.
- Para reduzir o custo relativo ao tempo de preparação das máquinas injetoras, pode-se aumentar o tamanho dos lotes produzidos para cada peça encomendada.
- É preciso identificar os clientes que adquirem os mesmos produtos, para negociar prazos de entrega comuns e assim aumentar o tamanho dos lotes produzidos.
- Solução: construir um grafo com cada vértice representando um cliente e ligar com uma aresta os que adquirem os mesmos produtos.
- Um clique no grafo representa o conjunto de clientes que adquirem os mesmos produtos.

Prova do Teorema de Cook

- A prova, bastante longa, mostra como construir Q a partir de A e E.
- A expressão booleana Q é longa, mas pode ser construída em tempo polinomial no tamanho de E.
- Prova usa definição matemática da **Máquina de Turing não-determinista** (MTND), capaz de resolver qualquer problema em $\mathcal{NP}$.
  - incluindo uma descrição da máquina e de como instruções são executadas em termos de fórmulas booleanas.
- Estabelece uma correspondência entre todo problema em $\mathcal{NP}$ (expresso por um programa na MTnd) e alguma instância de SAT.
- Uma instância de SAT corresponde à tradução do programa em uma fórmula booleana.
- A solução de SAT corresponde à simulação da máquina executando o programa em cima da fórmula obtida, o que produz uma solução para uma instância do problema inicial dado.

Teorema de Cook

- Existe algum problema em $\mathcal{NP}$ tal que se ele for mostrado estar em $\mathcal{P}$, implicaria $\mathcal{P} = \mathcal{NP}$?
- **Teorema de Cook:** Satisfabilidade (SAT) está em $\mathcal{P}$ se e somente se $\mathcal{P} = \mathcal{NP}$.
- Ou seja, se existisse um algoritmo polinomial determinista para *satisfabilidade*, então todos os problemas em $\mathcal{NP}$ poderiam ser resolvidos em tempo polinomial.
- A prova considera os dois sentidos:
  1. SAT está em $\mathcal{NP}$ (basta apresentar um algoritmo não-determinista que execute em tempo polinomial). Logo, se $\mathcal{P} = \mathcal{NP}$, então SAT está em $\mathcal{P}$.
  2. Se SAT está em $\mathcal{P}$, então $\mathcal{P} = \mathcal{NP}$. A prova descreve como obter de qualquer algoritmo polinomial não determinista de decisão A, com entrada E, uma fórmula $Q(A, E)$ de modo que Q é *satisfatível* se e somente se A termina com sucesso para E. O comprimento e tempo para construir Q é $O(p^3(n)\log(n))$, onde n é o tamanho de E e $p(n)$ é a complexidade de A.

Exemplo - Problema da Parada

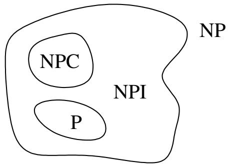
- É um exemplo de problema $\mathcal{NP}$-difícil que não é $\mathcal{NP}$-completo.
- Consiste em determinar, para um algoritmo determinista qualquer A com entrada de dados E, se o algoritmo A termina (ou entra em um *loop* infinito).
- É um problema **indecidível**. Não há algoritmo de qualquer complexidade para resolvê-lo.
- Mostrando que $SAT \propto$ problema da parada:
  - Considere o algoritmo A cuja entrada é uma expressão booleana na forma normal conjuntiva com n variáveis.
  - Basta tentar 2^n possibilidades e verificar se E é *satisfatível*.
  - Se for, A pára; senão, entra em *loop*.
  - Logo, o problema da parada é $\mathcal{NP}$-difícil, mas não é $\mathcal{NP}$-completo.

Problemas $\mathcal{NP}$-Completo e $\mathcal{NP}$-Difícil

- Um problema de decisão Π que seja $\mathcal{NP}$-difícil pode ser mostrado ser $\mathcal{NP}$-completo exibindo um algoritmo não-determinista polinomial para Π.
- Apenas problemas de decisão ("sim/não") podem ser $\mathcal{NP}$-completo.
- Problemas de otimização podem ser $\mathcal{NP}$-difícil, mas geralmente, se Π_1 é um problema de decisão e Π_2 um problema de otimização, é bem possível que $\Pi_1 \propto \Pi_2$.
- A dificuldade de um problema $\mathcal{NP}$-difícil não é menor do que a dificuldade de um problema $\mathcal{NP}$-completo.

Classe $\mathcal{NP}$-Intermediária

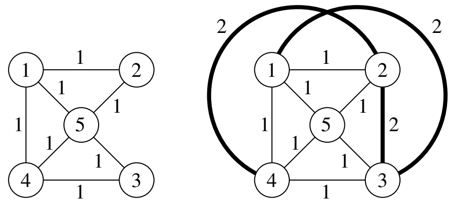
- Segunda descrição tentativa do mundo $\mathcal{NP}$, assumindo $\mathcal{P} \neq \mathcal{NP}$.



- Existe uma classe intermediária entre $\mathcal{P}$ e $\mathcal{NP}$ chamada $\mathcal{NPI}$.
- $\mathcal{NPI}$ seria constituída por problemas que ninguém conseguiu uma redução polinomial de um problema $\mathcal{NP}$-completo para eles, onde $\mathcal{NPI} = \mathcal{NP}$ - ($\mathcal{P} \cup \mathcal{NP}$-completo).

PCV é $\mathcal{NP}$-completo - Parte 2 da Prova

- Apresentar uma redução polinomial do ciclo de Hamilton para o PCV. Pode ser feita conforme o exemplo abaixo.



- Dado um grafo representando uma instância do ciclo de Hamilton, construa uma instância do PCV como se segue:
  1. Para cidades use os vértices.
  2. Para distâncias use 1 se existir um arco no grafo e 2 se não existir.
- A seguir, use o PCV para achar um roteiro menor ou igual a V.
- O roteiro é o ciclo de Hamilton.

PCV é $\mathcal{NP}$-completo - Parte 1 da Prova

- Mostrar que o Problema do Caixeiro Viajante (PCV) está em $\mathcal{NP}$.
- Prova a partir do problema **ciclo de Hamilton**, um dos primeiros que se provou ser $\mathcal{NP}$-completo.

```
procedure PCVND;
begin
  i := 1;
  for t := 1 to v do
    begin
    j := escolhe(i, lista-adj(i));
    antecessor[j] := i;
    end;
end;
```

- Isso pode ser feito:
  - apresentando (como abaixo) um algoritmo não-determinista polinomial para o PCV ou
  - mostrando que, a partir de uma dada solução para o PCV, esta pode ser verificada em tempo polinomial.

Prova de que um Problema é $\mathcal{NP}$-Completo

- São necessários os seguintes passos:
  1. Mostre que o problema está em $\mathcal{NP}$.
  2. Mostre que um problema $\mathcal{NP}$-completo conhecido pode ser polinomialmente transformado para ele.
- É possível porque Cook apresentou uma prova direta de que SAT é $\mathcal{NP}$-completo, além do fato de a redução polinomial ser transitiva ($SAT \propto \Pi_1$ & $\Pi_1 \propto \Pi_2 \Rightarrow SAT \propto \Pi_2$).
- Para ilustrar como um problema Π pode ser provado ser $\mathcal{NP}$-completo, basta considerar um problema já provado ser $\mathcal{NP}$-completo e apresentar uma redução polinomial desse problema para Π.

O Que Fazer para Resolver Problemas Exponenciais?

- Usar algoritmos exponenciais "eficientes" aplicando técnicas de tentativa e erro.
- Usar algoritmos aproximados. Acham uma resposta que pode não ser a solução ótima, mas é garantido ser próxima dela.
- Concentrar no caso médio. Buscar algoritmos melhores que outros neste quesito e que funcionem bem para as entradas de dados que ocorrem usualmente na prática.
  - Existem poucos algoritmos exponenciais que são muito úteis na prática.
  - Exemplo: Simplex (programação linear). Complexidade de tempo exponencial no pior caso, mas muito rápido na prática.
  - Tais exemplos são raros. A grande maioria dos algoritmos exponenciais conhecidos não é muito útil.

Problemas Exponenciais

- É desejável resolver instâncias grandes de problemas de otimização em tempo razoável.
- Os melhores algoritmos para problemas $\mathcal{NP}$-completo têm comportamento de pior caso exponencial no tamanho da entrada.
- Para um algoritmo que execute em tempo proporcional a 2^N, não é garantido obter resposta para todos os problemas de tamanho $N \geq 100$.
- Independente da velocidade do computador, ninguém poderia esperar por um algoritmo que leva 2^{100} passos para terminar sua tarefa.
- Um supercomputador poderia resolver um problema de tamanho $N = 50$ em 1 hora, ou $N = 51$ em 2 horas, ou $N = 59$ em um ano.
- Nem um computador paralelo com um milhão de processadores, (cada um sendo um milhão de vezes mais rápido que o mais rápido existente) é suficiente para $N = 100$.

Classe $\mathcal{NP}$-Completo - Resumo

- Problemas que pertencem a $\mathcal{NP}$, mas que podem ou não pertencer a $\mathcal{P}$.
- Propriedade: se qualquer problema $\mathcal{NP}$-completo puder ser resolvido em tempo polinomial por uma máquina determinista, então todos os problemas da classe podem, isto é, $\mathcal{P} = \mathcal{NP}$.
- A falha coletiva de todos os pesquisadores para encontrar algoritmos eficientes para estes problemas pode ser vista como uma dificuldade para provar que $\mathcal{P} = \mathcal{NP}$.
- Contribuição prática da teoria: fornece um mecanismo que permite descobrir se um novo problema é "fácil" ou "difícil".
- Se encontrarmos um algoritmo eficiente para o problema, então não há dificuldade. Senão, uma prova de que o problema é $\mathcal{NP}$-completo nos diz que o problema é tão "difícil" quanto todos os outros problemas "difíceis" da classe $\mathcal{NP}$-completo.

Membros Potenciais de $\mathcal{NPI}$

- **Isomorfismo de grafos:** Dados $G = (V, E)$ e $G' = (V, E')$, existe uma função $f : V \rightarrow V$, tal que $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$?
  - Isomorfismo é o problema de testar se dois grafos são o mesmo.
  - Suponha que seja dado um conjunto de grafos e que alguma operação tenha de ser realizada sobre cada grafo.
  - Se pudermos identificar quais grafos são duplicatas, eles poderiam ser descartados para evitar trabalho redundante.
- **Números compostos:** Dado um inteiro positivo k, existem inteiros $m, n > 1$ tais que $k = mn$?
  - Princípio da criptografia RSA: é fácil encontrar números primos grandes, mas difícil fatorar o produto de dois deles.

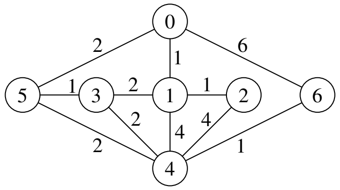
Encontrar um Ciclo de Hamilton em um Grafo Tentativa e Erro

```
procedure Dfs (Grafo: TipoGrafo);
var Tempo, i: integer;
    d: array[0..MAXNUMVERTICES] of integer;
  procedure Visita (k:integer);
  var j: integer;
  begin Tempo := Tempo + 1; d[k]  := Tempo;
        for j := 0 to Grafo.NumVertices − 1 do
          if Grafo.Mat[k, j] > 0
          then if d[j] = 0 then Visita (j);
  end;
begin
 Tempo := 0;
 for i := 0 to Grafo.NumVertices−1 do d[i] := 0;
 i := 0;
 Visita (i);
end;
```

- Obter algoritmo tentativa e erro a partir de algoritmo para caminhamento em um grafo.
- O Dfs faz uma busca em profundidade no grafo em tempo $O(|V| + |A|)$.

Ciclo de Hamilton - Tentativa e Erro

- Aplicando o Dfs ao grafo da figura abaixo a partir do vértice 0, o procedimento Visita obtém o caminho 0 1 2 4 3 5 6, o que não é um ciclo simples.



	1	2	3	4	5	6
0	1				2	6
1		1	2	4		
2				4		
3				2	1	
4					2	1
5						

- Para encontrar um ciclo de Hamilton, caso exista, devemos visitar os vértices do grafo de outras maneiras.
- A rigor, o melhor algoritmo conhecido resolve o problema tentando todos os caminhos possíveis.

Ciclo de Hamilton - Tentando Todas as Possibilidades

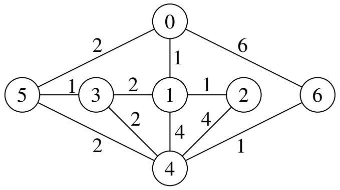
- Para tentar todas as possibilidades, vamos alterar o procedimento Visita.
- Desmarca o vértice já visitado no caminho anterior e permite que seja visitado novamente em outra tentativa.

```
procedure Visita (k:integer);
var j: integer;
begin
  Tempo := Tempo + 1; d[k]  := Tempo;
  for j := 0 to Grafo.NumVertices − 1 do
    if Grafo.Mat[k, j] > 0
    then if d[j] = 0
         then Visita (j);
  Tempo := Tempo − 1;
  d[k] := 0;
end;
```

- O custo é proporcional ao número de chamadas para o procedimento Visita.
- Para um grafo completo, (arestas ligando todos os pares de nós) existem $N!$ ciclos simples. Custo é proibitivo.

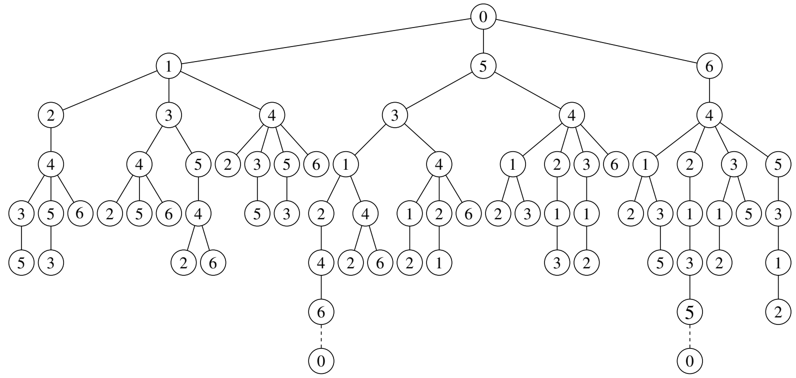
Ciclo de Hamilton - Tentando Todas as Possibilidades

- Para o grafo



Ciclo de Hamilton - Tentando Todas as Possibilidades

A árvore de caminhamento é:



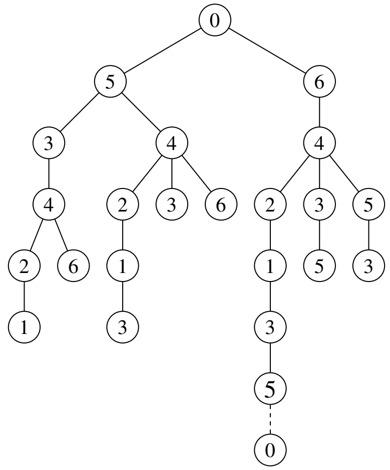
- Existem duas respostas: 0 5 3 1 2 4 6 0 e 0 6 4 2 1 3 5 0.

Ciclo de Hamilton - Tentativa e Erro com Poda

- Diminuir número de chamadas a Visita fazendo "**poda**" na árvore de caminhamento.
- No exemplo anterior, cada ciclo é obtido duas vezes, caminhando em ambas as direções.
- Insistindo que o nó 2 apareça antes do 0 e do 1, não precisamos chamar Visita para o nó 1 a não ser que o nó 2 já esteja no caminho.

Ciclo de Hamilton - Tentativa e Erro com Poda

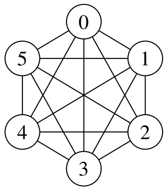
- Árvore de caminhamento obtida:



Ciclo de Hamilton - Tentativa e Erro com Poda

- Entretanto, esta técnica não é sempre possível de ser aplicada.
- Suponha que se queira um caminho de custo mínimo que não seja um ciclo e passe por todos os vértices: 0 6 4 5 3 1 2 é solução.
- Neste caso, a técnica de eliminar simetrias não funciona porque não sabemos *a priori* se um caminho leva a um ciclo ou não.

Heurística para o PCV



	1	2	3	4	5
0	3	10	11	7	25
1		8	12	9	26
2			9	4	20
3				5	15
4					18

- Caminho ótimo para esta instância: 0 1 2 5 3 4 0 (comprimento 58).
- Para a heurística do vizinho mais próximo, se iniciarmos pelo vértice 0, o vértice mais próximo é o 1 com distância 3.
- A partir do 1, o mais próximo é o 2, a partir do 2 o mais próximo é o 4, a partir do 4 o mais próximo é o 3, a partir do 3 restam o 5 e o 0.
- O comprimento do caminho 0 1 2 4 3 5 0 é 60.

Heurística para o PCV

- Algoritmo do vizinho mais próximo, heurística gulosa simples:
  1. Inicie com um vértice arbitrário.
  2. Procure o vértice mais próximo do último vértice adicionado que não esteja no caminho e adicione ao caminho a aresta que liga esses dois vértices.
  3. Quando todos os vértices estiverem no caminho, adicione uma aresta conectando o vértice inicial e o último vértice adicionado.
- Complexidade: $O(n^2)$, sendo n o número de cidades, ou $O(d)$, sendo d o conjunto de distâncias entre cidades.
- Aspecto negativo: embora todas as arestas escolhidas sejam localmente mínimas, a aresta final pode ser bastante longa.

Heurísticas para Problemas $\mathcal{NP}$-Completo

- **Heurística**: algoritmo que pode produzir um bom resultado (ou até a solução ótima), mas pode também não obter solução ou obter uma distante da ótima.
- Uma heurística pode ser determinista ou probabilística.
- Pode haver instâncias em que uma heurística (probabilística ou não) nunca vai encontrar uma solução.
- A principal diferença entre uma heurística probabilística e um **algoritmo Monte Carlo** é que o algoritmo Monte Carlo tem que encontrar uma solução correta com uma certa probabilidade (de preferência alta) para qualquer instância do problema.

Ciclo de Hamilton - *Branch-and-Bound*

- Outra saída para tentar diminuir o número de chamadas a Visita é por meio da técnica de ***branch-and-bound***.
- A ideia é cortar a pesquisa tão logo se saiba que não levará a uma solução.
- Corta chamadas a Visita tão logo se chegue a um custo para qualquer caminho que seja maior que um caminho solução já obtido.
- Exemplo: encontrando 0 5 3 1 2 4 6, de custo 11, não faz sentido continuar no caminho 0 6 4 1, de custo 11 também.
- Neste caso, podemos evitar chamadas a Visita se o custo do caminho corrente for maior ou igual ao melhor caminho obtido até o momento.

Algoritmos Aproximados - Definição

- Um algoritmo aproximado para um problema Π é um algoritmo polinomial que produz uma solução $S(I)$ para uma instância I de Π.
- O comportamento do algoritmo A é descrito pela **razão de aproximação**

$$R_A(I) = \frac{S(I)}{S^*(I)},$$

  que representa um problema de minimização
- No caso de um problema de maximização, a razão é invertida.
- Em ambos os casos, $R_A(I) \geq 1$.

Medindo a Qualidade da Aproximação

- O comportamento de algoritmos aproximados quanto Ă qualidade dos resultados (não o tempo para obtê-los) tem de ser monitorado.
- Seja I uma instância de um problema Π e seja $S^*(I)$ o valor da solução ótima para I.
- Um algoritmo aproximado gera uma solução possível para I cujo valor $S(I)$ é maior (pior) do que o valor ótimo $S^*(I)$.
- Dependendo do problema, a solução a ser obtida pode minimizar ou maximizar $S(I)$.
- Para o PCV, podemos estar interessados em um algoritmo aproximado que minimize $S(I)$: obtém o valor mais próximo possível de $S^*(I)$.
- No caso de o algoritmo aproximado obter a solução ótima, então $S(I) = S^*(I)$.

Algoritmos Aproximados para Problemas $\mathcal{NP}$-Completo

- Para projetar algoritmos polinomiais para “resolver” um problema de otimização $\mathcal{NP}$-completo é necessário relaxar o significado de resolver.
- Removemos a exigência de que o algoritmo tenha sempre de obter a solução ótima.
- Procuramos algoritmos eficientes que não garantem obter a solução ótima, mas sempre obtêm uma próxima da ótima.
- Tal solução, com valor próximo da ótima, é chamada de solução aproximada.
- Um **algoritmo aproximado** para um problema Π é um algoritmo que gera **soluções aproximadas** para Π.
- Para ser útil, é importante obter um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado.

Heurística para o PCV

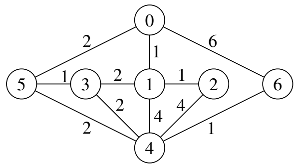
- Embora o algoritmo do vizinho mais próximo não encontre a solução ótima, a obtida está bem próxima do ótimo.
- Entretanto, é possível encontrar instâncias em que a solução obtida pode ser muito ruim.
- Pode mesmo ser arbitrariamente ruim, uma vez que a aresta final pode ser muito longa.
- É possível achar um algoritmo que garanta encontrar uma solução que seja razoavelmente boa no pior caso, desde que a classe de instâncias consideradas seja restrita.

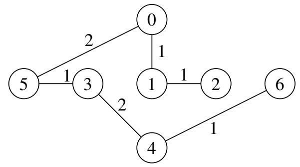
Limite Inferior para a Solução do PCV a Partir da AGM

- A partir da AGM, podemos derivar o limite inferior para o PCV.
- Considere uma aresta (x_1, x_2) do caminho ótimo do PCV. Remova a aresta e ache um caminho iniciando em x_1 e terminando em x_2.
- Ao retirar uma aresta do caminho ótimo, temos uma árvore geradora que consiste de um caminho que visita todas as cidades.
- Logo, o caminho ótimo para o PCV é necessariamente maior do que o comprimento da AGM.
- O **limite inferior** para o custo deste caminho é a AGM.
- Logo, $Otimo_{PCV} > AGM$.

Árvore Geradora Mínima (AGM)

- Considere um grafo $G = (V, A)$, sendo V as n cidades e A as distâncias entre cidades.
- Uma árvore geradora é uma coleção de $n - 1$ arestas que ligam todas as cidades por meio de um subgrafo conectado único.
- A **árvore geradora mínima** é a árvore geradora de custo mínimo.
- Existem algoritmos polinomiais de custo $O(|A| \log |V|)$ para obter a árvore geradora mínima quando o grafo de entrada é dado na forma de uma matriz de adjacência.
- Grafo e árvore geradora mínima correspondente:





Algoritmos Aproximados para o PCV

- Quando o problema exige distâncias não restritas à desigualdade triangular, basta adicionar uma constante k a cada distância.
- Exemplo: as três distâncias envolvidas são 2, 3 e 10, que não obedecem à desigualdade triangular pois $2 + 3 < 10$. Adicionando $k = 10$ às três distâncias obtendo 12, 13 e 20, que agora satisfazem a desigualdade triangular.
- O problema alterado terá a mesma solução ótima que o problema anterior, apenas com o comprimento da rota ótima diferindo de $n \times k$.
- Cabe observar que o PCV equivale a encontrar no grafo $G = (V, A)$ um **ciclo de Hamilton** de custo mínimo.

Algoritmos Aproximados para o PCV

- Seja $G = (V, A)$ um grafo não direcionado, completo, especificado por um par (N, d).
- N é o conjunto de vértices do grafo (cidades), e d é uma função distância que mapeia as arestas em números reais, onde d satisfaz:
  1. $d(i, j) = d(j, i)\ \forall i, j \in N$,
  2. $d(i, j) > 0\ \forall i, j \in N$,
  3. $d(i, j) + d(j, k) \geq d(i, k)\ \forall i, j, k \in N$
- 1ª propriedade: a distância da cidade i até outra adjacente j é igual à de j até i.
- Quando isso não acontece, temos o problema conhecido como **PCV Assimétrico**
- 2ª propriedade: apenas distâncias positivas.
- 3ª propriedade: **desigualdade triangular**. A distância de i até j somada com a de j até k deve ser maior do que a distância de i até k.

Limite Superior de Aproximação para o PCV

- A desigualdade triangular permite utilizar a AGM para obter um **limite superior** para a razão de aproximação com relação ao comprimento do caminho ótimo.
- Vamos considerar um algoritmo que visita todas as cidades, mas pode usar somente as arestas da AGM.
- Uma possibilidade é iniciar em um vértice folha e usar a seguinte estratégia:
  - Se houver aresta ainda não visitada saindo do vértice corrente, siga aquela aresta para um novo vértice.
  - Se todas as arestas a partir do vértice corrente tiverem sido visitadas, volte para o vértice adjacente pela aresta pela qual o vértice corrente foi inicialmente alcançado.
  - Termine quando retornar ao vértice inicial.

Limite Superior de Aproximação para o PCV - Busca em Profundidade

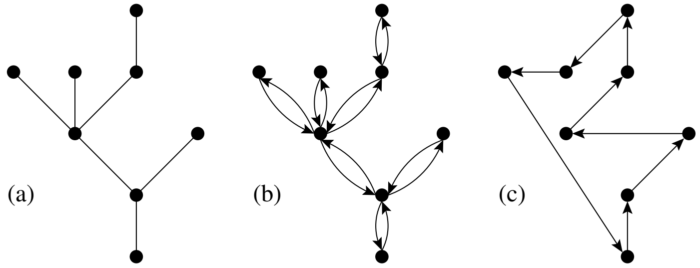
- O algoritmo descrito anteriormente é a Busca em Profundidade aplicada à AGM.
- Verifica-se que:
  - o algoritmo visita todos os vértices.
  - nenhuma aresta é visitada mais do que duas vezes.
- Obtém um caminho que visita todas as cidades cujo custo é menor ou igual a duas vezes o custo da árvore geradora mínima.
- Como o caminho ótimo é maior do que o custo da AGM, então o caminho obtido é no máximo duas vezes o custo do caminho ótimo. $Caminho_{PCV} < 2Otimo_{PCV}$.
- Restrição: algumas cidades são visitadas mais de uma vez.
- Para contornar o problema, usamos a desigualdade triangular.

Limite Superior de Aproximação para o PCV - Desigualdade Triangular

- Introduzimos curto-circuitos que nunca aumentam o comprimento total do caminho.
- Inicie em uma folha da AGM, mas sempre que a busca em profundidade for voltar para uma cidade já visitada, salte para a próxima ainda não visitada.
- A rota direta não é maior do que a anterior indireta, em razão da desigualdade triangular.

Limite Superior de Aproximação para o PCV - Desigualdade Triangular

- Se todas as cidades tiverem sido visitadas, volte para o ponto de partida.



- O algoritmo constrói um caminho solução para o PCV porque cada cidade é visitada apenas uma vez, exceto a cidade de partida.

Limite Superior de Aproximação para o PCV - Desigualdade Triangular

- O caminho obtido não é maior que o caminho obtido em uma busca em profundidade, cujo comprimento é no máximo duas vezes o do caminho ótimo.
- Os principais passos do algoritmo são:
  1. Obtenha a árvore geradora mínima para o conjunto de n cidades, com custo $O(n^2)$.
  2. Aplique a busca em profundidade na AGM obtida com custo $O(n)$:
     - Inicie em uma folha (grau 1).
     - Siga uma aresta não utilizada.
     - Se for retornar para uma cidade já visitada, salte para a próxima ainda não visitada (rota direta menor que a indireta pela desigualdade triangular).
     - Se todas as cidades tiverem sido visitadas, volte à cidade de origem.
- Assim, obtivemos um algoritmo polinomial de custo $O(n^2)$, com uma razão de aproximação garantida para o pior caso de $R_A \leq 2$.

Como Melhorar o Limite Superior a Partir da AGM

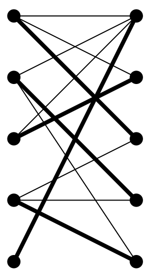
- No algoritmo anterior um caminho para o caixeiro viajante pode ser obtido dobrando os arcos da AGM, o que leva a um pior caso para a razão de aproximação no máximo igual a 2.
- Melhora-se a garantia de um fator 2 para o pior caso, utilizando o conceito de grafo Euleriano.
- Um **grafo Euleriano** é um grafo conectado no qual todo vértice tem grau par.
- Um grafo Euleriano possui um **caminho Euleriano**, um ciclo que passa por todas as arestas exatamente uma vez.
- O caminho Euleriano em um grafo Euleriano, pode ser obtido em tempo $O(n)$, usando a busca em profundidade.
- Podemos obter um caminho para o PCV a partir de uma AGM, usando o caminho Euleriano e a técnica de curto-circuito.

Como Melhorar o Limite Superior a Partir da AGM

- Passos do algoritmo:
  - Suponha uma AGM que tenha cidades do PCV como vértices.
  - Dobre suas arestas para obter um grafo Euleriano.
  - Encontre um caminho Euleriano para esse grafo.
  - Converta-o em um caminho do caixeiro viajante usando curto-circuitos.
- Pela desigualdade triangular, o caminho do caixeiro viajante não pode ser mais longo do que o caminho Euleriano e, conseqüentemente, de comprimento no máximo duas vezes o comprimento da AGM.

Casamento Mínimo com Pesos

- Christophides propôs uma melhoria no algoritmo anterior utilizando o conceito de **casamento mínimo com pesos** em grafos.
- Dado um conjunto contendo um número par de cidades, um casamento é uma coleção de arestas M tal que cada cidade é a extremidade de exatamente um arco em M.



- Um casamento mínimo é aquele para o qual o comprimento total das arestas é mínimo.
- Todo vértice é parte de exatamente uma aresta do conjunto M.
- Pode ser encontrado com custo $O(n^3)$.

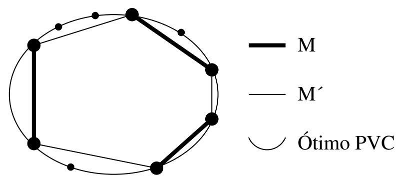
Casamento Mínimo com Pesos

- Seja I uma instância do PCV, e $Comp(T)$, $Comp(M)$ e $Comp(M')$, respectivamente, a soma dos comprimentos de T, M e M'.
- Pela desigualdade triangular devemos ter que: $Comp(M) + Comp(M') \leq Otimo(I)$,
- Assim, ou M ou M' têm de ter comprimento menor ou igual a $Otimo(I)/2$.
- Logo, o comprimento de um casamento mínimo para os vértices de grau ímpar de T tem também de ter comprimento no máximo $Otimo(I)/2$.
- Desde que o comprimento de M é menor do que o caminho do caixeiro viajante ótimo, podemos concluir que o comprimento do grafo Euleriano construído é:

$$Comp(I) < \frac{3}{2} Otimo(I).$$

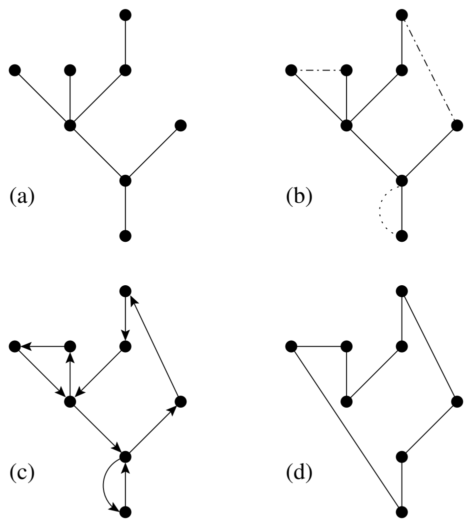
Casamento Mínimo com Pesos

- Basta agora determinar o comprimento do grafo de Euler.
- Caminho do caixeiro viajante em que podem ser vistas seis cidades correspondentes aos vértices de grau ímpar enfatizadas.



- O caminho determina os casamentos M e M'.

Casamento Mínimo com Pesos - Exemplo



a. Uma árvore geradora mínima T.

b. T mais um casamento mínimo dos vértices de grau ímpar.

c. Caminho de Euler em (b).

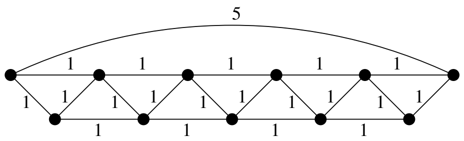
d. Busca em profundidade com curto-circuito.

Casamento Mínimo com Pesos

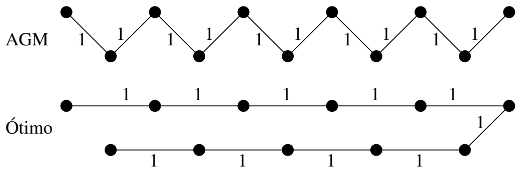
- Considere a AGM T de um grafo.
- Alguns vértices em T já possuem grau par, assim não precisariam receber mais arestas se quisermos transformar a árvore em um grafo Euleriano.
- Os únicos vértices com que temos de nos preocupar são os vértices de grau ímpar.
- Existe sempre um número par de vértices de grau ímpar, desde que a soma dos graus de todos os vértices tenha de ser par porque cada aresta é contada exatamente uma vez.
- Uma maneira de construir um grafo Euleriano que inclua T é simplesmente obter um casamento para os vértices de grau ímpar.
- Isto aumenta de um o grau de cada vértice de grau ímpar. Os de de grau par não mudam.
- Se adicionamos em T um casamento mínimo para os vértices de grau ímpar, obtemos um grafo Euleriano que tem comprimento mínimo dentre aqueles que contêm T.

Algoritmo de Christophides - Pior Caso

- Exemplo de pior caso do algoritmo de Christofides:



- A AGM e o caminho ótimo são:



- Neste caso, para uma instância I:

$$C(I) = \frac{3}{2}[Otimo(I) - 1],$$

onde o $Otimo(I) = 11$, $C(I) = 15$, e $AGM = 10$.

Casamento Mínimo com Pesos - Algoritmo de Christophides

- Os principais passos do algoritmo de Christophides são:
  1. Obtenha a AGM T para o conjunto de n cidades, com custo $O(n^2)$.
  2. Construa um casamento mínimo M para o conjunto de vértices de grau ímpar em T, com custo $O(n^3)$.
  3. Encontre um caminho de Euler para o grafo Euleriano obtido com a união de T e M, e converta o caminho de Euler em um caminho do caixeiro viajante usando curto-circuitos, com um custo de $O(N)$.
- Assim obtivemos um algoritmo polinomial de custo $O(n^3)$, com uma razão de aproximação garantida para o pior caso de $R_A < 3/2$.